(1)

Jointly-Owned Objects for Collaboration:

Operating-System Support and Protection Model

Sheng-Uei Guan
University of North Carolina at Chapel Hill

Hussein Abdel-Wahab
Old Dominion University

Peter Calingaert
University of North Carolina at Chapel Hill

N00014-86-K-0680
1989

Abstract

As *real-time collaboration* becomes more frequent, it is common for a group of users to create and own an object jointly. The use of multi-user tools makes the existence of *jointly-owned objects* a necessity: a participant who joins a multi-user tool written by others knows that the user agent executed in his name is not a Trojan horse if the multi-user tool is jointly owned by all the participants. In this paper, we discuss the requirements and issues behind jointly-owned objects. By generalizing these requirements we have implemented a *conditionally jointly-owned object*. The conditions take the form of a quorum or a list of users who have the rights to access an object or to change its protection state. We sketch a design of conditionally jointly-owned objects, and apply the same concepts to subjects. *Authority-* and *quorum-based objects* are investigated as instances of conditionally jointly-owned objects. We show that conditionally jointly-owned objects can also be used to resolve the conflicts that may arise among joint owners. We generalize Graham and Denning's protection model to incorporate these jointly-owned entities. Operating system support for conditionally jointly-owned objects is specified at the system-call level. Examples are provided to demonstrate the usefulness of conditionally jointly-owned objects.

Keywords: computer-supported cooperative work, multi-user tool, jointly-owned object, jointly-owned subject, protection models

To appear in

The Journal of Systems and Software

1. INTRODUCTION

With the falling cost of hardware and communication, more and more people are experimenting today with computer-supported cooperative work (CSCW) [1,2]. We usually find CSCW in an environment with networked personal computers, workstations, and multi-user computer systems. Here are some sample applications of this technology: accountant and client at different locations each looking at the same spreadsheet on their personal computers; two researchers jointly writing a paper over a wide-area network; an application programmer and a system programmer collaborating to find an elusive bug in a program execution for which both share the same view of the process output.

Conceptually, the object being worked on in a CSCW scenario is owned jointly by the collaborators. Such joint ownership is common in real life, as for joint bank accounts and condominium real estate. In computer systems, however, an object is usually owned by a single user. The operating-system support provided for such objects does not serve the conceptual requirements of CSCW. We seek therefore to introduce into operating systems the concepts and mechanisms needed to support joint ownership of objects.

Another requirement for joint ownership arises from security concerns. A multi-user cooperation tool when executed usually creates, for each participant, a user agent that runs under that participant's domain. Some multi-user tools will be written by system programmers, and some by the collaborators themselves. For the former, tools installed will be trusted by the collaborators and used without any security concern, just as users usually trust system utilities. For the latter, a participant who joins a multi-user tool written by others has no way of knowing that the user agent executed in his name is not a Trojan horse [3]. (A Trojan horse usually refers to a trapdoor, in a program, that allows unauthorized access to other objects.) This doubt can be relieved when a collaborator knows about the multi-user tool he joins. But it cannot be totally removed unless he is sure that the multi-user tool he knows and trusts cannot be replaced unless he is notified. This requirement cannot be fulfilled with traditional singly-owned objects, since one of the authors of the multi-user tool usually has full ownership of the tool and has rights to make such a change.

There exist, to be sure, a few examples of jointly-owned objects in computer systems: a virtual circuit that, once established, each party can read from, write into, or disconnect; a link in a hypertext environment that spans across nodes in files of two users, each of whom jointly owns the link and can delete it whenever appropriate; a multi-threaded task [4], described more fully in Section 2.3. Nevertheless, these objects are highly specialized and by themselves inadequate to support CSCW.

In Shamir's work [5], a joint-encryption scheme allows several users to share a secret object. No single user can open the object unless a certain number of users present keys to open it. Although such a scheme seems to provide functions of jointly-owned objects, there is a difference. A jointly-encrypted object is usually not owned by the whole group; it is owned by one group member. Hence the owner can change its protection mode. It is difficult also to change the set of members who share a jointly-encrypted object. The object must be re-encrypted with another key and the members informed of the new key. The maintenance of the key by each member is a burden, and there is always a possibility that a member forgets his key or exposes it accidentally.

Inasmuch as singly-owned objects are managed by operating systems, it is appealing to integrate the jointly-owned objects with the singly-owned objects. The problem of supporting jointly-owned objects by using a mechanism designed for the support of singly-owned objects is that when an object is created, one member of the group is overly trusted with full ownership. Even if the group decides that the object is read-only, the assigned owner can still change its

protection mode and write over it. Deletion of the single-owner from the group requires a reassignment of all the jointly-owned objects to other users in the group.

What should be the protection model for jointly-owned objects? Although many protection models have appeared in the literature [6-10], none has dealt with this possibility. In the following, we first summarize Graham and Denning's protection model [7]. To help resolve conflicts among joint-owners, we generalize jointly-owned objects to conditionally jointly-owned objects, and present a mechanism for realizing them. We extend the Graham and Denning model to provide a protection basis for conditionally jointly-owned objects and subjects, and specify operating-system support for conditionally jointly-owned objects at the system-call level. Finally, we present a functional description and system-call level interface of the design of a jointly-owned subject, *i.e.* the multi-user process [11], and provide examples. Details of our implementation have been described in Guan's dissertation [11].

2. CONDITIONALLY JOINTLY-OWNED OBJECTS

2.1 Definitions

First we give some definitions. An *object* is an entity to which access must be controlled. A *subject* is what Graham and Denning call an "active entity" (*i.e.* a process) whose access to objects must be controlled. A subject may create an object, and becomes the *owner* of the object. Every subject is also an object, because it must be protected.

We propose a mechanism to allow multiple owners to specify some *condition* to the system. A condition defines one or more subsets of the set of users who have rights to an object. It can be a quorum (*e.g.* presence of at least two joint-owners), an authority-list (*e.g.* presence of joint-owners A and B), or something more complex, (*e.g.* presence of more than 60% of the joint-owners, including A). We call these objects *conditionally jointly-owned* (CJO) objects. The system ensures that the condition is met before the object can be accessed or its protection state changed.

An object's condition has two distinct parts. An *access-condition* (AC), if placed on an object, must be met before the owners or authorized users can access the object. A *control-condition* (CC), if placed on an object, must be met before the owners or authorized users can change the protection state of the object (*e.g.* grant an access right to another user, destroy the object). Each user (process) when making an access or changing the protection state of an object needs to inform the system whether a joint action is intended. If so, the system will wait until the required number of participants join and then verify that the condition is met. Using the access- or control-condition, the joint-owners' conflicts are resolved with their joint presence.

An access condition is useful if the joint-owners of an object want more awareness of each other's actions on the object. For example, when two users jointly open a bank safe, they are aware of each other's actions on the safe in addition to the knowledge of joint presence. An access condition can include presence constraints for read, write, or execute access that require all or a majority of the joint-owners to be present.

A jointly-owned (JO) object is a special case of CJO object with null control-condition. Each owner of a JO object has full ownership. The access-condition of a JO object may be non-null. Obviously, a singly-owned object is a special case of a jointly-owned object.

2.2 Operations on CJO Objects

Creation. A CJO object may be created by several users jointly, *e.g.* through a *multi-user process*, as described in [11]. A multi-user process is a process jointly owned by all attaching users. It provides multiple terminal interfaces to attaching users. The creator of a process makes it multi-user by giving a list of users who may join. When a participant joins the process, he becomes a joint-owner of the process. Standard input, output, and error channels are created for this user, whose terminal becomes attached to the channels When an object is created by a multi-user process, the participants become joint-owners of the CJO object. During the creation of the object, the users specify the access- and control-condition jointly.

Alternatively, the owner of an object may grant ownership to another user, if that user accepts. In most protection systems, granting an access right needs no agreement of the grantee [7]. Our model requires that the grantee agree. This is because ownership frequently implies obligation. Sometimes a user does not want such a granted ownership; he may even be charged for disk space if he jointly owns an object. The original owner of an object makes it jointly owned by specifying the joint-owners. The granting of ownership to a user is completed when that user accepts it, thereby becoming *committed*. A *uncommitted* user has no owner right to the object.

There is one technical difficulty in requiring a user to commit before ownership is granted. The protection state of a CJO object may not even be changed until all the joint-owners commit, because the control-condition is not met until then. To solve this difficulty, we define the *effective control-condition* (ECC) as the restriction of the specified control-condition (CC) to the set of committed joint owners. For example, let the CC be the presence of any three joint-owners, including Smith. If only two joint-owners have committed, and Smith has not, then the ECC is the presence of those two joint-owners. Once three or more joint-owners, including Smith, have committed, the ECC becomes equal to the specified CC. The ECC, when different from the specified CC, is used in place of the specified CC. Similarly, we define the *effective access-condition* (EAC) with respect to the specified access condition (AC).

Validation of access. One difficulty of validating access to a CJO object can readily be seen. With computer access, users need not even gather together physically to access an object jointly. With single-user processes, it is difficult for users to provide evidence to the system that they are indeed "together" to open the object. If each user issues open in his process, the requests received by the operating system are still serialized, and the system has no way to verify that users are together. The system cannot simply wait until all users have issued their requests.

Assume that users with different interests are collaborating in subgroups on different sections of an object. The system needs to know whether the requests issued from the users' processes are related. For example, assume that a quorum-based object has four users who have read access rights, and a read-quorum equal to two. Suppose the users form two groups. The read requests from these two groups of users should not be correlated by the system because they may work on different parts of a document.

The difficulty can be solved with a multi-user process, described further in Section 5. The multi-user process can be programmed to ask agreement from its participants and perform the joint action for the users. The multi-user process notifies each participant of the result of the joint action by replicating it to each participant's standard output channel. Alternatively, the difficulty can be solved in the following way: before accessing a CJO object jointly, one user process provides to the system some information (*e.g. time_out* or the number of users to be together) and asks the system to return an unforgeable token. It then distributes the token to its cooperating user processes that want to access this object jointly. These user processes may notify the attaching users, seek their agreement, and present the token when making their requests, so that the system knows that they are together to make the access. The system waits until all expected participants

make the access request (or the specified *time_out* expires). It then checks whether the effective access- or control-condition is met.

With a multi-user process, a joint operation is performed in a straightforward manner. A **read** or **write** action is performed once; the result is returned to the multi-user process itself. With several processes issuing a joint operation through a token, we have "write once, read many" operation. A write operation, whether into a file, channel, data structure, or memory, is performed only once. Thus for a joint-write operation, only one process, preferably the one that asks the system to assign a token for the joint operation, needs to tell the system all the information needed for the write operation. For a read operation, whether from a file, channel, data structure, or memory, the result is replicated to all the participating processes. Thus all the participating processes need to provide the system consistent information regarding how the read operation is to be done (*e.g.* how many bytes to read, where to store the result).

Miscellaneous. When the ECC is null, a joint-owner may withdraw his ownership at will. Otherwise, the ECC must be met. A withdrawing user is removed from the committed users list.

The access- or control-condition of a CJO object can be changed if the issuing user(s) meet the effective control-condition.

A CJO object can be deleted upon command of the joint-owners. The effective control-condition must be met. When the condition is null, this can be done by any joint-owner. Of course, an object is removed when its last joint-owner withdraws.

2.3 Jointly-Owned Subjects

Because subjects are also considered to be objects, it is natural to expect that the concept of "jointly-owned" can be applied to subjects. The *multi-threaded task* [4] is an example. (A task is an execution environment — chiefly an address space and a protection state — in which threads may run; a thread is the basic unit of CPU utilization, roughly equivalent to an independent program counter.) All threads within a multi-threaded task execute in parallel and share the same address space and capabilities. Any thread may suspend, resume, or destroy the task as a whole. Thus the task is jointly owned by its threads.

A jointly-owned (JO) subject is defined as a subject that has several owners, each of whom has full ownership. A subject can be jointly created and owned by several owners; alternatively, an owner can grant ownership to another subject, who becomes a joint-owner if he agrees. With this extension, ownership can be granted, and the relation "owner" no longer defines a tree hierarchy. A joint-owner cannot invalidate the ownership of another joint-owner. Thus ownership, once granted, cannot be revoked. A joint-owner (subject) may grant some of its rights to a JO subject; the conferred rights or the subject itself may be removed by another joint-owner with appropriate rights. An object that is created by a JO subject is a JO object. The notion of "conditionally jointly-owned" can be applied to subjects.

The multi-user process is another JO subject. The process is jointly owned by all attaching participants. An object or a process created during the execution of a multi-user process will normally be owned by the joint-owners.

3. PROTECTION MODELS

3.1 Review of Graham and Denning's Model

To permit the cooperation of mutually suspicious subsystems, Graham and Denning [7] proposed a protection model based on Lampson's work [6]. They left the case of "jointly-owned" unsolved [7-10]. We chose their model for extension because it incorporates the widely used access matrix [10,12]. We summarize their model here and encourage our readers to study their original paper.

There are three components in their model: *objects*, *subjects*, and *rules*. Objects and subjects are as defined in Section 2. A unique identifier is assigned to each object. When a subject creates an object, the system grants to the subject the "owner" right to the object. The owner right allows the subject to grant to itself any access right to the object. When the object being created is also a subject, the creator grants to it a "control" right. This right allows the subject to read or delete rights from its protection state.

The information specifying the types of access that subjects have to objects can be represented conceptually as an access matrix A, with subjects identifying the rows and objects the columns. Element A[S,X] of the matrix specifies the access rights held by subject S to object X.

A *copy flag* can be associated with an access right. If the copy flag is on, the subject may grant to any other subject that access right to the object. If the copy flag is off, the subject may not grant such access. Here, the copy flag is an asterisk.

A *monitor* exists for each type of object; it validates all accesses to objects of that type. An access proceeds as follows:

1. S initiates access to X in manner *a*, *e.g.* read, write, *etc.*
2. The computer system supplies the triple (S, *a*, X) to the monitor of X.
3. The monitor of X interrogates the access matrix to determine whether *a* is in A[S,X]. If so, access is permitted; otherwise, it is denied.

Rules control the making of changes in the protection state. Each rule has three parts: a command issued by the subject, an authorization that must be satisfied, and the operation that results if the subject has the required authorization. Graham and Denning stated eight rules (Table I of [7]). The rules are enforced by an *access-matrix monitor*.

Graham and Denning make a restriction that each subject is owned or controlled by at most one other subject. Enforcing this maintains a tree hierarchy of objects. It is still possible in their model for the owner right of a non-subject object to be granted, but they argued that either multiple ownership should not be provided or coordination among the joint-owners themselves needs to be done to avoid contradictory actions, *e.g.* one joint-owner grants access the others do not want granted.

3.2 Our Extended Model

Access. We extend the protection model by associating with each CJO object two fields in the access matrix: access-condition (AC), and control-condition (CC). The effective access-condition (EAC) is AC evaluated without uncommitted joint-owners. The following notations are adopted in the access matrix:

owner& ; *uncommitted joint-owner*
owner ; *owner* or *committed joint-owner*

Generally, an access proceeds as follows:

1. Access to an object X in manner *a* is initiated by a conceptual subject SV. Formally, SV (for "subject vector") is a single- or multi-component vector whose elements are the individual subjects.
2. The system supplies the triple (SV, *a*, X) to the monitor of X.
3. The monitor of X interrogates the access matrix to determine whether *a* is in A[SV,X] and the effective access-condition is satisfied. If *both* conditions are satisfied, access is permitted; otherwise, it is denied.

For authority-based objects, the last rule says that the effective access-authority, *i.e.* the access-authority members who have committed, must be present to make access. For users to access a quorum-based object, |SV| (dimension of SV) may not be less than the effective access-quorum.

An example: shown in the access matrix of Figure 1 is a CJO object X with three joint-owners B, C, D and one user E. It is an authority- and quorum-based object with access-condition {read-quorum = 0, write-quorum = 2, execute-quorum = 0}, and control-condition {control-authority = B, control-quorum = 2}. Since all joint-owners are committed, the effective access/control-condition is the specified access/control-condition. Because the read- and execute-quorums are zero, user B, C, D or E can read or execute the object individually. Any two users together are allowed to write the object. Users C and D together, although they are joint-owners and meet the control-quorum, may not change the protection state of X because the control-authority (owner B) is not present. Users B and C together, or B and D together, are allowed to change its protection state because the control-authority is present and the control-quorum is met.

It should be noted here that for multiple owners J,K of an object X, their access matrix entries A[J,X] and A[K,X] may not always be identical, because an owner can delete rights (see next section) from its own entry. As shown in Figure 1, the access- and control-condition are stored with an object. An ideal place to store them is with the access control list of the object if there is one; otherwise they may be stored as part of the features of the object.

Protection System Commands. Protection system commands are listed in Figure 2, which is in two ways an extension of Table I of [7]. For the convenience of readers who are familiar with the Graham and Denning rules, we have numbered our first eight to correspond with theirs. Our first extension is to add to their rules R1-R8 the concepts of the various conditions and to impose some necessary restrictions. The second extension is the introduction of three new commands. Rules R9 and R10 govern new commands for adding and removing joint-owners. Rule R11 provides for changing AC or CC. In Figure 2, SV can stand for a single subject or several subjects, but we restrict the grantee S to a single subject.

Modifications to Graham and Denning's Rules. Rule R1 allows one or several subjects (SV) to transfer (replicate) an access right held for an object X to any other subject, if the copy flag ("*") is set and the corresponding effective control-condition is met. Note that a subject does not need to be an owner to be able to transfer a right. Transferring a right to another subject results in a protection state change of the object X, because the grantee will then be able to access the object. Hence the corresponding ECC must be met.

Rule R2 allows one or several joint-owners to grant to any other subject access rights for an owned object, if the ECC is met. Rule R3 allows one or several subjects to delete from any other subject access rights for an object if they have *control* right over that *subject*, or *owner* right over that *object*. For both cases the ECC must be met. The right *a* granted, transferred, or deleted in rules R1-R3 is restricted not to be "owner" or "owner&". This is because granting or

withdrawing ownership needs special handling. Rules R9 and R10 are provided for those operations.

We digress at this point to exemplify the differences between our rules and those of Graham and Denning, by citing their rule R3. Their authorization is either 'control' in $A[S_0,S]$ or 'owner' in $A[S_0,X]$, where S_0 is the subject issuing the command; their operation is to delete 'a' from $A[S,X]$. We augmented their authorization by adding the ECC; we restricted their operation by excluding ownership as a right deletable by this rule.

Rule R4 allows one or more subjects SV to read the protection state of another subject S regarding an object X, if SV has control right to subject S, or SV has owner right to object X.

Rule R5 allows one or several subjects jointly to create a CJO object by issuing the **create** command. The subjects are considered to be committed immediately, *i.e.* access right *owner* is stored into each joint-owner's matrix entry. Creating a CJO subject (rule R7) is like creating a CJO object, except that the subject created will have a control right to itself (as in Graham and Denning's model). Only the owner or joint-owners are able to destroy a CJO object or subject; the ECC must be met (rules R6, R8).

The Added Rules. Another way to create a CJO object is through converting a singly-owned object, *i.e.* through granting ownership: **add_joint** (rule R9). Note that **add_joint** is used either for an owner or joint-owners to add a new joint-owner or for a user to commit himself as a joint-owner. When a new joint-owner S is added for an object X, the access right *owner&* is inserted into A[S,X]. When that user commits as a joint-owner, his access right *owner&* is changed into *owner*.

Rule R10 is used for joint-owners to remove an uncommitted or committed joint-owner, if the ECC is met. It is possible for a joint-owner to withdraw unilaterally only if the ECC so allows. Rule R11 allows joint-owners to change the access- or control-condition of an object if the ECC is met. In this rule, *NC* stands for the new condition, and *a/c* is a flag specifying whether *NC* is an access- or control-condition.

Because ownership cannot be transferred or granted using rule R1 or R2, and **add_joint** requires a grantee to commit before he accepts the ownership, it is not possible to transfer or grant ownership without the grantee's agreement. Similarly, because ownership cannot be deleted using rule R3 and **withdraw_joint** requires the ECC to be met for a joint-owner to withdraw, it is not possible in general for a joint-owner to withdraw unilaterally. We now explain these commands for authority- and quorum-based objects.

To convert a singly-owned object into an authority-based object, **change_condition** can be issued to specify the access- and control-conditions: an access-authority-list and a control-authority-list are specified. **Add_joint** is then issued to specify an owners-list. To commit, a user issues **add_joint** and he must be on the owners-list. To add a new joint-owner, **add_joint** is again issued, and the effective control-authority members must be among the issuing users SV. A joint-owner can withdraw his joint-ownership if the effective control-authority-list is null; otherwise the members of the effective control-authority list must be present.

A quorum-based object can be created similarly by converting a singly-owned object, or it can be created by several subjects specifying access-quorums and the control-quorum with the **create** command. These subjects become the joint-owners. Adding or withdrawing a joint-owner requires that the number of issuing joint-owners be not less than the effective control-quorum. Changing the access-quorums or the control-quorum can be done through **change_condition**, which requires the effective control-quorum to be met.

Note that the sequence of issuing **change_condition** and **add_joint** to convert a singly-owned object into CJO object is significant. If **add_joint** is issued first, there is a possibility that, before any condition on the presence of joint-owners is imposed, one joint-owner commits and

changes the protection state before the original owner can specify a CC on it. An alternative would be to have a separate protection system command **make_joint** to allow an owner to specify the list of joint-owners and the conditions simultaneously. To add joint-owners, or to commit, **add_joint** is used. To change the conditions, **change_condition** is used. Another possibility is that when a control-condition is not specified the default is to require all joint-owners to be present.

In our extended model, granting ownership needs the grantee's agreement. With a slight modification, however, ownership can be granted without the grantee's agreement. (Rules R1, R2, R3, and R10 are changed by removing the term "owner&"; rule R9 is changed by removing references to committing, and "owner" is stored in A[S,X] when ownership is granted.) Each grantee is then treated as immediately committed when ownership is granted.

Correctness and Trust. Graham and Denning use an informal approach to show the correctness of their model. To show the correctness of the extension, we follow their approach under the assumption that their original model is correct. As they state [7]:

> To prove that a protection model or its implementation is correct, one must show that a subject can never access an object except in an authorized manner. Two things must be proved: (I) any action by a subject that does not change the protection state cannot be an unauthorized access; and (II) any action by a subject that does change the protection state cannot lead to a new protection state in which some subject has unauthorized access to some object.

Regarding the first (I), the extended model allows the additional possibility that several subjects jointly access an object. Would this lead to an unauthorized access? The answer is no, as the validation against A[SV,X] or A[SV,S] is still performed for each participating subject. It is assumed here that the system associates with each subject a unforgeable identifier, that each monitor acts correctly, *i.e.* it interrogates the correct entry in the access matrix for each access, and that no monitor except the access-matrix monitor is able to change the contents of the access matrix. Any action done is based on the joint-owners' pre-agreed control-condition, and hence is authorized.

Regarding the second (II), let's look at rules R1-R8 first. Except for rules R4, R5, and R7, the others are extended with the additional ECC restriction, which means the system will perform additional checking. For example, in Graham and Denning's model, it is possible to grant or transfer ownership to another user. When multiple ownerships arise in this case, they have no control when conflicting actions among multiple owners are issued, *e.g.* through rules R1-R3. With the extended model, the ECC terms imply that additional checking is to be done before a user who has control or owner right can change the protection state. The new protection states reached through these rules (R1-R8) are the same as in Graham and Denning's model except for rules R5 and R7, where additionally AC and CC are stored with X. This new protection state does not result in a new state in which some subject gains unauthorized access to some object, because AC and CC are conditions to be met (not rights).

Consider now rules R9-R11. Rule R9 allows a new owner to be added; the resulting new protection state allows that new owner access to the object X if he commits. As the application of rule R9 is based on trust, *i.e.* an owner (or several owners) trusts a new user to own his (their) object, this user is authorized to have the access. So the new protection state does not result in a new state in which some subject has unauthorized access to some object. It is obvious that rule R10 does not result in a new state in which some subject has unauthorized access to some object. In rule R11, the multiple owners agree to make a change of AC or CC, either allowing owners more freedom or reducing owners' freedom. Either case results in authorized access only.

4. IMPLEMENTATION OF CJO OBJECTS

4.1 System Call Interface

To create authority- or quorum-based objects, we can specify an *authority-list* or *quorums* for each kind of access right (*e.g.* read, write, execute, or control). The number of users on the authority-list can be one, several, or zero. In the last case no authority presence is required to control protection state changes on the object. The authority list must be a subset of the owners. The *control-quorum* specifies the number of owners needed to change an owner, authority, quorum, or protection mode. The *effective control-quorum* is the lesser of the specified control-quorum (as specified in the **make_joint** call) and the current number of committed joint-owners. The effective control-quorum is validated in the protection state change on a jointly-owned object.

To implement CJO objects, we created six system calls in the C language for use in a Unix environment.

make_joint *(object_name, owners_list, authority_list, control_quorum, read_quorum, write_quorum, execute_quorum)*

```
char *object_name;
char *owners_list;
char *authority_list;
        authority_list is applied with the control_quorum
int control_quorum, read_quorum, write_quorum, execute_quorum;
```

To simplify our design, the above *authority_list* does not include the access-authority list. The extension to one with access-authority list is straightforward. Note that the specified quorums should be nonnegative.

add_joint *(object_name, joint_owner_name)*

```
char *object_name, *joint_owner_name;
```

If the issuer is a joint-owner who hasn't committed, this call makes him a joint-owner. This call also allows a user to be granted joint-ownership of an object by several joint-owners when the effective control-authority and control-quorum are met.

withdraw_joint *(object_name, joint_owner_name)*

```
char *object_name, *joint_owner_name;
```

For an owner to withdraw, the effective control-authority and control-quorum must be met. If a withdrawing user is one of the control-authority, his name is removed from the *authority_list*. Three other calls are used to change the control-authority field or quorums of an authority- or quorum-based object. The effective control-authority and control-quorum must be met.

add_authority *(object_name, joint_owner_name)*

```
char *object_name, *joint_owner_name;
```

withdraw_authority *(object_name, joint_owner_name)*
char *object_name, *joint_owner_name;

change_quorum(*object_name, crwx_flag, new_quorum*)
char *object_name,
char crwx_flag; *'c' for changing the control_quorum,*
'r' for changing the read_quorum,
'w' for changing the write_quorum,
'x' for changing the execute_quorum.
int new_quorum;

4.2 Examples.

Let's see how the mechanisms proposed in Section 4.1 can be applied to several instances of conditionally jointly-owned objects. Although the first two examples come from the real world, we believe that they can be modeled and realized in the computer world with the introduction of CJO objects.

Example 8.1 — Joint Account. Suppose a couple John and Mary open a joint account Y:

make_joint ("Y", "john mary", "", 1, 1, 1, 1)

The call can be issued through a multi-user process with John and Mary as owners. This account Y is a JO object. Either John or Mary can get information from, deposit into, withdraw from, or delete the account.

Example 8.2 — Contract. Suppose user Guan makes a contract proposal of object X and presents it to user Chen, who agrees by committing to it. User Guan issues the following:

make_joint ("X", "guan chen", "guan chen", 2, 1, 2, 1)

The contract is proposed to be a CJO object owned by both users, controllable and writable only when both Guan and Chen are present. It is readable by any joint-owner. User Chen, after seeing the contract proposal, agrees by issuing the following:

add_joint ("X", "chen")

Note that specifying the *control_quorum* and *write_quorum* as 2 helps to resolve any possible conflict between Guan and Chen because a joint action is needed before the contract can be changed or deleted.

Example 8.3 — Multi-User Tool (Solving the Trojan Horse Problem). The following example shows how the CJO concepts can be applied to the computer world: a multi-user tool is made jointly owned to solve the Trojan horse problem.

A multi-user tool can be implemented in two ways: centralized with a server synchronizing messages or resolving contending accesses to objects, or decentralized with all user agents coordinating and synchronizing in a distributed manner. A user usually has a user agent running under his domain. The user agent program is usually the same for each participant and can be made jointly owned by all of them. The same can be done with the server program. For example, suppose that a multi-user tool has two components, a user agent program X and a server program Y, and is used by users Guan, Wang, and David. User Guan writes the tool, and proposes it be jointly owned by three of them through issuing:

make_joint ("X", "guan wang david", "guan wang david", 3, 1, 3, 1)

make_joint ("Y", "guan wang david", "guan wang david", 3, 1, 3, 1)

This says that rewriting the contents of the multi-user tool or changing the protection state of the multi-user tool requires the presence of all three owners. Users Wang and David, after committing with **add_joint**, will be sure that no change (either to the contents or to the write-quorum or authority-list of the tool) can be made without their presence.

5. MULTI-USER PROCESS

The multi-user process mechanism supports multi-user tool development and sharing of user privileges in real-time cooperation. Traditionally, a process is associated with a single user. For real-time cooperation, we propose the multi-user process for *access control list systems* and *systems with mixed strategy* [3]. A system with mixed strategy is a system that uses an access control list for the secondary storage or file system, while using a capability scheme for the rest; the capabilities cannot be copied into the file system.

A process runs initially with its creator as the owner (against whom the protection check is made). The creator makes the process "multi-user" by issuing **allow_join**, which specifies a nickname and a list of users who may join. The creator then issues **wait_join** when ready to accept participants to join. A process that wishes to join issues **join_proc**.

When a user on the list specified by the **allow_join** call issues **join_proc** from a single-user process, that process is suspended. Standard input, output, and error channels are created in the multi-user process for this user, whose terminal becomes attached to the channels. The user becomes *active* in the multi-user process. The multi-user process can read input from the joining user's standard input channel and can write output to his standard output channel. An active user leaves the multi-user process by issuing an exit control signal from his terminal or when the process terminates. The process can be killed by an active user with a special control signal.

allow_join *(nickname, users_list)*

```
char *nickname;
char *users_list;
```

If a multi-user process with the same name and created by the same creator already exists, the call returns ERROR. The creator is an assumed participant whose name need not be specified in the *users_list*. If *users_list* is omitted, any eligible user on this machine can join.

wait_join *(ifnotjoin, time_out, joint_user)*

```
int ifnotjoin, time_out;
JOIN_INFO *joint_user;

typedef struct j_user {
  char *username;
  int in, out, err;
  } JOIN_INFO;
```

Wait_join waits for one user (process) at a time to join. A program can be coded with a simple loop so that **wait_join** is executed several times until all the users on the *users_list* join. The flag *ifnotjoin* selects among three options. (1) The calling process will block until the process

of a user on the specified list issues **join_proc**. (2) The calling process will block until either that event occurs or the specified *time_out* interval elapses. (3) The calling process will not block.

With this call, the multi-user process is ready to accept participants. The returned information *joint_user* includes the name of the joining user and the standard input, output, and error descriptors created for his terminal.

join_proc *(nickname, creator_name, ifnotexist, time_out)*

```
char *nickname, *creator_name;
int ifnotexist, time_out;
```

To avoid naming conflicts, a joining participant is required to specify the name of the multi-user process creator in addition to the *nickname*. The flag *ifnotexist* selects among the same three actions as before if the multi-user process does not exist or is not ready to accept participants.

A multi-user process is jointly owned by its active users, against whom the protection check is made. If an active user leaves by issuing an exit control signal, that user loses his ownership to the multi-user process and his original process is then resumed.

A multi-user process achieves a shared *workspace*. This is an abstraction that denotes a collection of objects belonging to some cooperative work and the software tools needed to access these objects. Each participant has in front of him the shared workspace where he can operate with some tools on the same objects that other participants see. For example, researchers writing a joint paper will have in their workspace objects such as sections, figures and tables, and tools such as editors, formatters, and spelling checkers. The resource of a multi-user process participant can be shared whenever the process opens it or acquires a capability for it. Simultaneous manipulation of objects across multiple user domains (*e.g.* a process simultaneously opening objects under different users' domains) is possible because the process runs under the union of multiple user domains. A departing user may leave behind capabilities for his objects so that others can continue working on them.

How is it possible that a multi-user process runs with multiple user privileges? We assume that participants in a multi-user process will share with each other the access rights needed for object access when they issue **join_proc**. Thus, when a multi-user process accesses an object, the user(s) who have the access right grant it to the others so that they can make joint access. The grantor(s) must have the copy flag set with their rights, and the effective control-condition must be met. The right granted will be used only for the life of the multi-user process. We see an analogy in real-time collaboration, where we allow a participant to share access to an object. After the collaboration, the participant may no longer access the object.

Implementation details and examples demonstrating the usefulness of the multi-user process have been described in [11].

6. CONCLUSION

Effective sharing of workspace objects is a fundamental issue in multi-user cooperative work. The proposed CJO mechanism serves as an extension to traditional singly-owned objects, allowing sharing of ownership and trusted use of multi-user collaboration tools. The multi-user process, as a special case of CJO object, allows the implementation of a fully shared workspace.

A design of conditionally jointly-owned (CJO) objects has been laid out and a protection basis provided for it. The proposed jointly-owned object mechanism extends the functions of

operating systems. Ownership can indeed be shared. Although granting ownership in our extended protection model needs the grantee's agreement, only slight changes are required to make ownership grantable without the grantee's agreement.

Access rights can also be shared. Conflicts among joint-owners can be solved with CJO objects: actions must be approved with an access- or control-condition on the presence of the owners. Access- and control-conditions are provided as mechanisms for specifying presence conditions; they are designed as general mechanisms so that different user policy decisions can be implemented.

Two varieties of CJO objects have been provided: authority- and quorum-based objects. The former gives the presence condition by a user list, the latter by a count. Operating-system support for these CJO objects is specified at the system-call level. The problem of using a multi-user cooperation tool is also solved by making the tool a CJO object: by specifying a write-access-condition and control-condition that require the collaborators' presence. A user can trust a multi-user cooperation tool he uses since he knows that neither the contents nor the write-access-condition of the tool can be changed without his being notified.

We have tested the concepts presented in this paper by implementing, under the 4.3BSD Unix operating system [13], a library of system call interface routines and server to support conditionally jointly-owned objects and multi-user processes, as well as two concepts not discussed in this article: shared capability lists and dynamic groups with shared viewing. We have written several programs based on the library calls to demonstrate the feasibility of our approach. Among these programs is a recoding of the Remote Shared Workspace application reported in [2]. This application permits multiple, mutually remote users to share a single-user tool for collaborative tasks such as editing. Our programs have demonstrated the utility of our concepts; greater execution efficiency will await a direct implementation in the operating-system kernel.

Increasing productivity has been a major goal of this century. Facilitating cooperative work using computers will increase productivity. Incorporating more support into operating systems facilitates cooperative work and the development of more productive cooperation tools. It remains to be seen how much closer this will bring us to the goal.

ACKNOWLEDGEMENTS

The authors wish to thank their colleague Don Smith for valuable comments. This work was partially supported by the Office of Naval Research, under contract N00014-86-K-0680, and by IBM, under Shared University Research Agreement #826.

REFERENCES

[1] Irene Greif, *Computer-Supported Cooperative Work: A Book of Readings*, Morgan Kaufmann, Palo Alto, California, 1988.

[2] H. M. Abdel-Wahab, S.-U. Guan, and J. Nievergelt, Shared Workspaces for Group Collaboration: An Experiment using Internet and UNIX Interprocess Communications, IEEE *Communications* 26, 10-16, 1988.

[3] J. H. Saltzer and M. D. Schroeder, The Protection of Information in Computer Systems, *Proceedings of the IEEE* 63, 1278-1308, 1975.

[4] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young, Mach: A New Kernel Foundation for UNIX Development, *USENIX Winter Conference Proceedings*, 93-112, 1985.

[5] A. Shamir, How to Share a Secret, *CACM* 22, 612-613, 1979.

[6] B. Lampson, Protection, *Proceedings, Fifth Annual Princeton Conference on Information Sciences and Systems*, Department of Electrical Engineering, Princeton University, Princeton, New Jersey, 437-443, 1971; reprinted in *ACM Operating Systems Review* 8, 18-24, 1974.

[7] G. S. Graham and P. J. Denning, Protection — Principles and Practice, *AFIPS Conference Proceedings* 40, 417-429, 1972.

[8] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, Protection in Operating Systems, *CACM* 19, 461-471, 1976.

[9] T. A. Linden, Operating System Structures to Support Security and Reliable Software, *ACM Computing Surveys* 8, 409-445, 1976.

[10] C. E. Landwehr, Formal Models for Computer Security, *ACM Computing Surveys* 13, 247-275, 1981.

[11] S.-U. Guan, A Model, Architecture, and Operating System Support for Shared Workspace Cooperation, Ph.D. Dissertation, University of North Carolina at Chapel Hill, 1989.

[12] M. Maekawa, A. E. Oldehoeft, and R. R. Oldehoeft, Operating Systems: Advanced Concepts, Benjamin/Cummings, Menlo Park, California, 1987.

[13] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, Reading, Massachusetts, 1989.